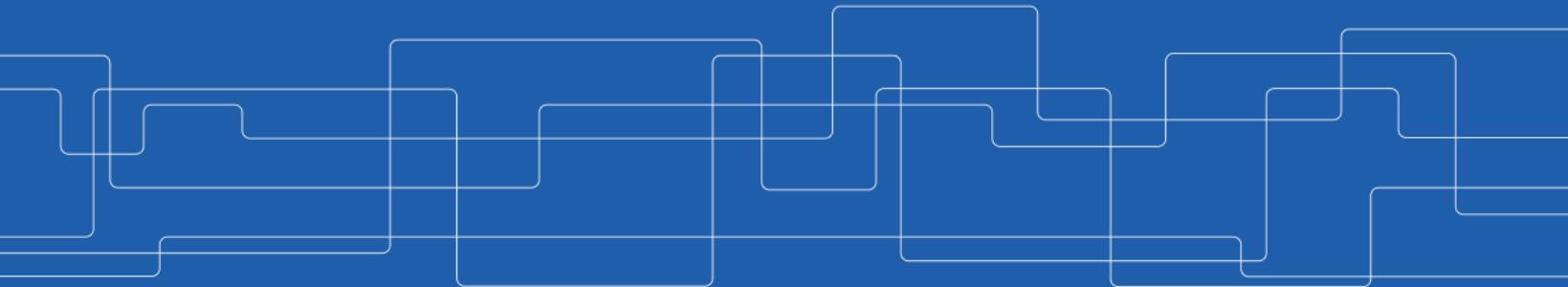




Distributed Learning

Amir H. Payberah
payberah@kth.se
2020-12-08



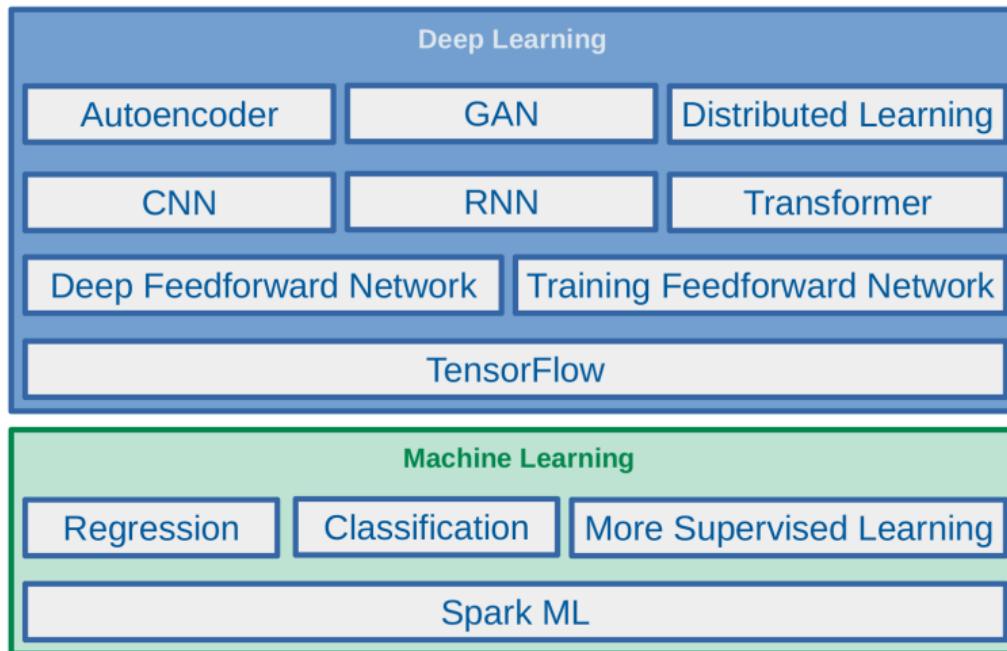


The Course Web Page

<https://id2223kth.github.io>
<https://tinyurl.com/y6kcpmzy>

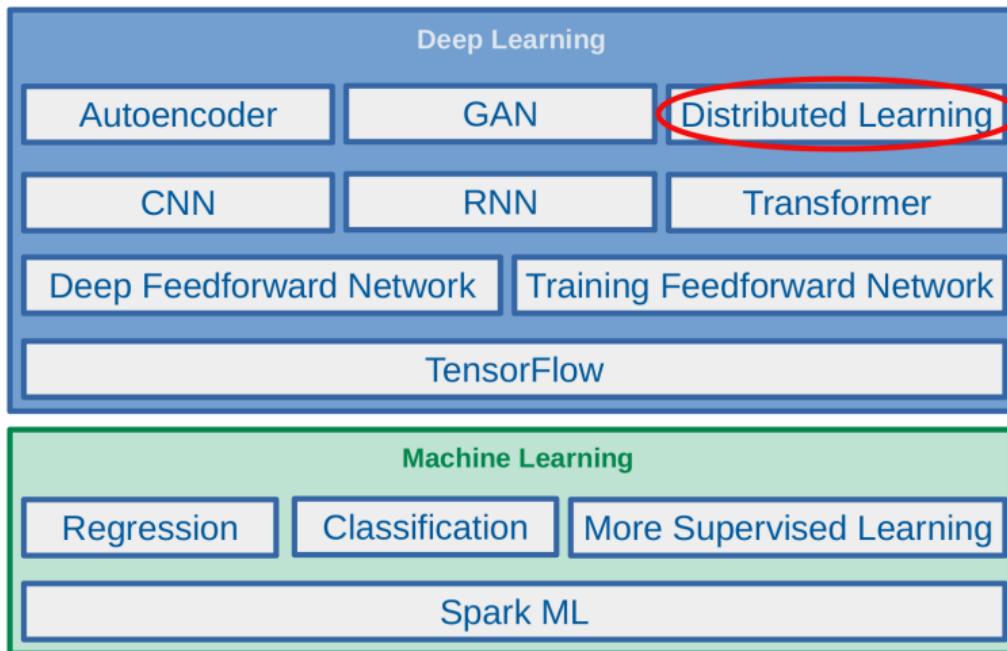


Where Are We?



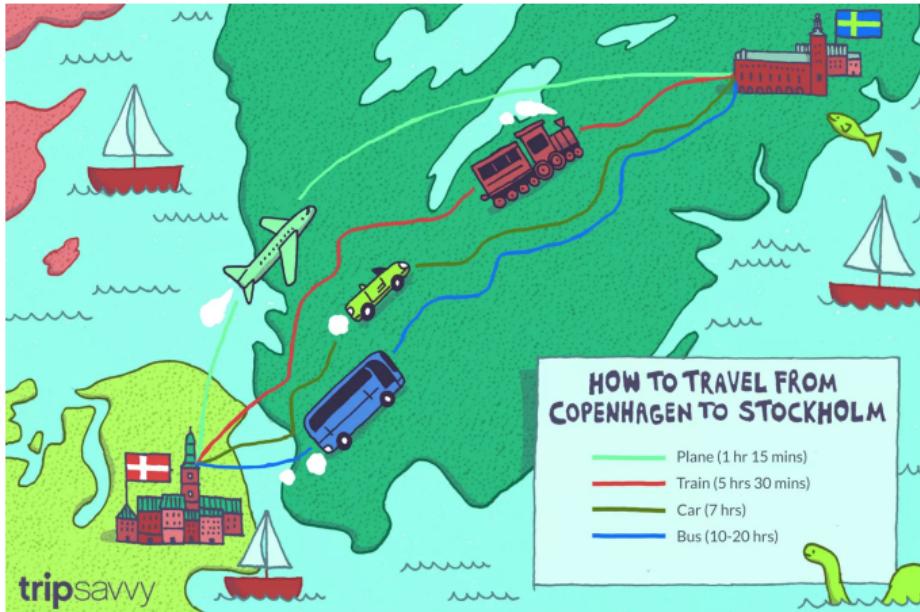


Where Are We?





A few Words about CPU and GPU



[<https://www.tripsavvy.com/how-to-get-from-copenhagen-to-stockholm-1626275>]

Ferrari or Truck?



Ferrari or Truck?

- ▶ Pick up your partner?
- ▶ Moving the furniture?

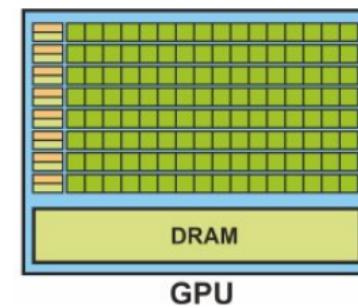
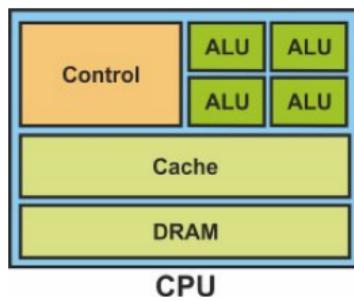


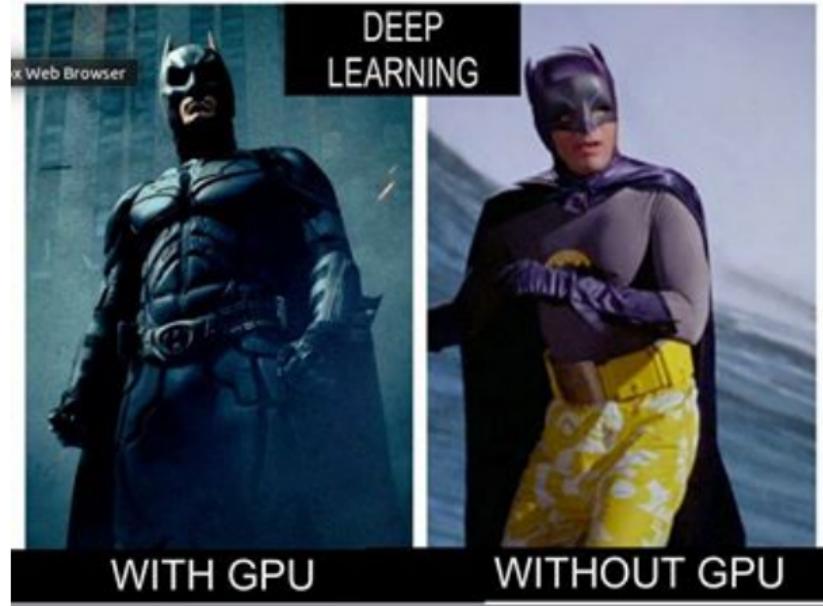


CPU vs GPU



CPU vs GPU







Do We Need GPU for Deep Learning?

- ▶ Which components of a DNN would require intense hardware resource?
- ▶ A few candidates are:
 - Preprocessing input data
 - Training the model
 - Storing the trained model
 - Deployment of the model

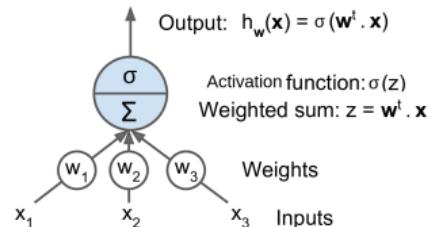


- ▶ Which components of a DNN would require intense hardware resource?
- ▶ A few candidates are:
 - Preprocessing input data
 - Training the model
 - Storing the trained model
 - Deployment of the model

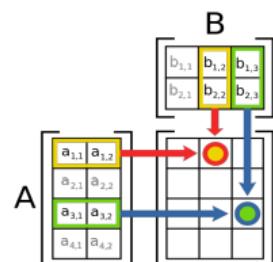


Training a Model

- ▶ Forward pass: input is passed through the DNN and an output is generated.
- ▶ Backward pass: weights are updated on the basis of error we get in forward pass.



- ▶ Both of these operations are essentially **matrix multiplications**.





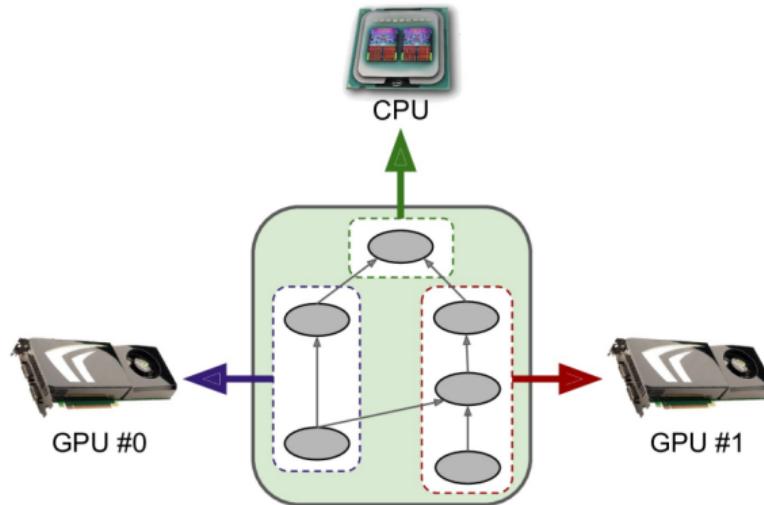
How to Train a Model Faster?

- ▶ The computationally intensive part of neural network is made up of multiple matrix multiplications.
- ▶ How can we make it faster?
- ▶ Do these operations at the same time, instead of doing it one after the other.
- ▶ This is in a nutshell why we use GPU instead of a CPU for training a neural network.



Placing Operations and Variables on Devices (1/4)

- ▶ For now, let's assume to run everything on a single machine.





Placing Operations and Variables on Devices (2/4)

- ▶ Place the **data preprocessing** operations on CPUs, and the **NN operations** on GPUs.
- ▶ Adding more **CPU RAM** to a machine is **simple and cheap**, whereas the **GPU RAM** is an **expensive and limited** resource.
 - If a variable is not needed in the next few training steps, it should probably be placed on the CPU (e.g., **datasets generally belong on the CPU**).
- ▶ **GPUs** usually have a fairly **limited communication bandwidth**, so it is important to avoid **unnecessary data transfers** in and out of the GPUs.



Placing Operations and Variables on Devices (3/4)

- ▶ By default, all **variables/operations** are placed on the **first GPU**: `/gpu:0`.
- ▶ Variables/operations that **do not have a GPU kernel** are placed on the **CPU**: `/cpu:0`.
- ▶ A **kernel** is a **variable or operation's implementation** for a specific data and device type.
 - For example, there is a GPU kernel for the `float32 tf.matmul()` operation, but there is no GPU kernel for `int32 tf.matmul()` (only a CPU kernel).



Placing Operations and Variables on Devices (4/4)

- ▶ TensorFlow automatically decides which device to execute an operation and copies tensors to that device.
- ▶ However, TensorFlow operations can be explicitly placed on specific devices using the `tf.device` context manager.



Manual Device Placement (1/3)

- ▶ Use `with tf.device` to create a **device context**.
- ▶ All the **operations within that context** will run on the **same designated device**.

```
tf.debugging.set_log_device_placement(True)

a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
c = tf.matmul(a, b)
print(c)
```

Output:

```
Executing op MatMul in device /job:localhost/replica:0/task:0/device:GPU:0
tf.Tensor(
[[22. 28.]
 [49. 64.]], shape=(2, 2), dtype=float32)
```



Manual Device Placement (2/3)

```
tf.debugging.set_log_device_placement(True)

with tf.device('/cpu:0'):
    a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
    b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])

c = tf.matmul(a, b)
print(c)
```

```
Executing op MatMul in device /job:localhost/replica:0/task:0/device:GPU:0
tf.Tensor(
[[22. 28.
[49. 64.]], shape=(2, 2), dtype=float32)
```

- ▶ Here, `a` and `b` are assigned to `CPU:0`.
- ▶ Since a device was not explicitly specified for the `matmul` operation, it will be run on the default device `GPU:0`.



Manual Device Placement (3/3)

```
tf.debugging.set_log_device_placement(True)

with tf.device('/cpu:0'):
    a = tf.constant([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
    b = tf.constant([[1.0, 2.0], [3.0, 4.0], [5.0, 6.0]])
    c = tf.matmul(a, b)

print(c)
```

```
Executing op MatMul in device /job:localhost/replica:0/task:0/device:CPU:0
tf.Tensor(
[[22. 28.
[49. 64.]], shape=(2, 2), dtype=float32)
```



Parallel Execution Across Multiple Devices



Parallelization

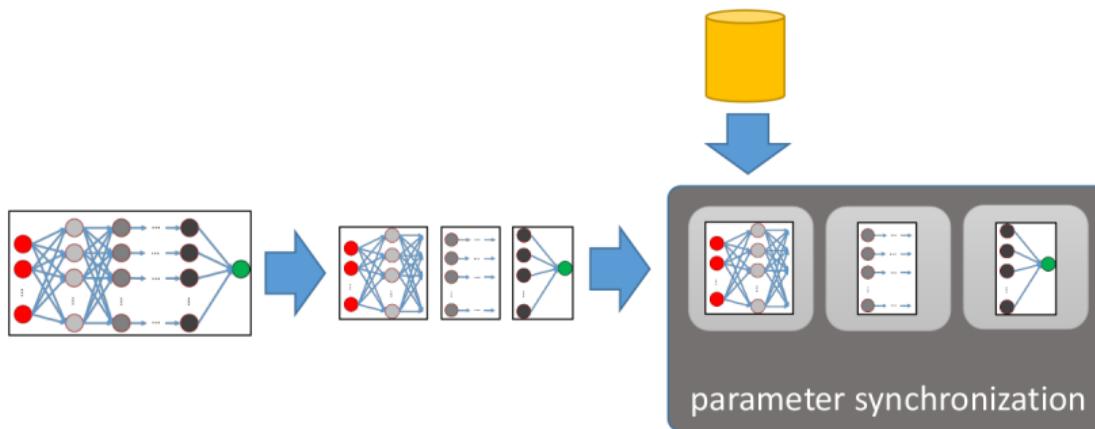
- ▶ Train **large deep learning models** with **huge amounts of training data**.
- ▶ **Parallelization** and **distribution** are essential.
- ▶ Two main approaches to training a single model across **multiple devices**:
 - **Model** parallelization
 - **Data** parallelization



Model Parallelization

Model Parallelization

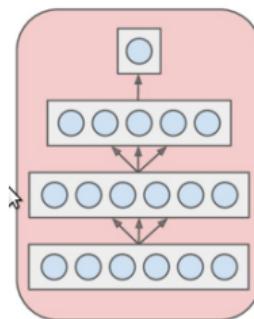
- ▶ The **model** is split across **multiple devices**.
- ▶ Depends on the **architecture** of the **NN**.



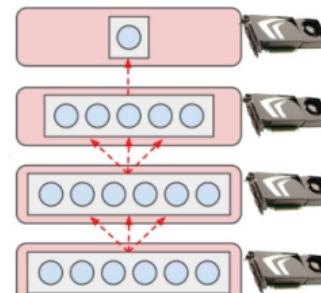
[Mayer, R. et al., arXiv:1903.11314, 2019]

Fully Connected Model Parallelization (1/2)

- ▶ To place **each layer** on a different device.
- ▶ **Not good:** each layer needs to **wait** for the output of the **previous layer** before it can do anything.



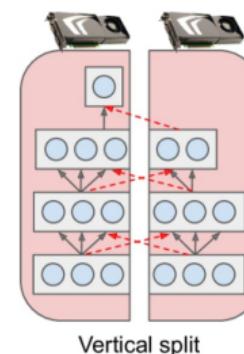
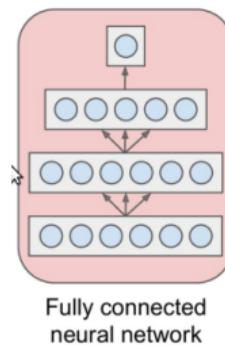
Fully connected
neural network



One layer per device

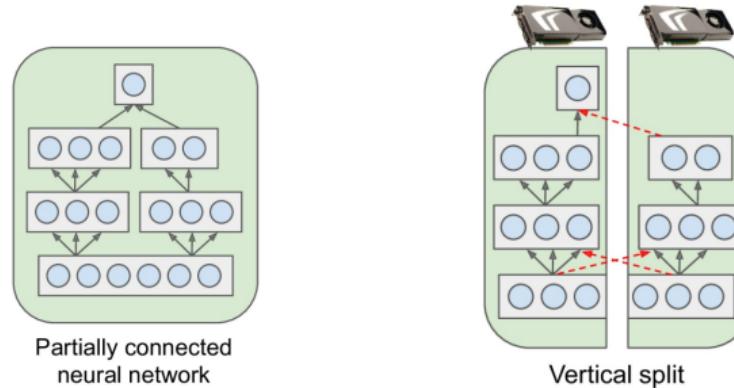
Fully Connected Model Parallelization (2/2)

- ▶ Slice the model **vertically**.
 - E.g., the **left half** of each layer on one device, and the **right part** on another device.
- ▶ **Slightly better:** both halves of each layer can indeed work **in parallel**.
- ▶ Each half of the next layer requires the **output of both halves**: lot of **cross-device communication**.



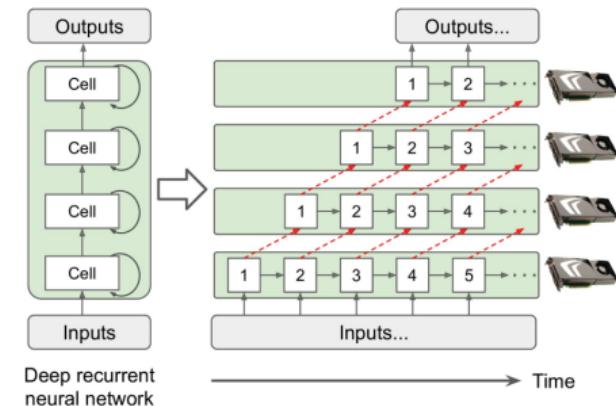
CNN Model Parallelization

- ▶ Some NN, such as CNN, contains layers that are only **partially connected** to the **lower layers**.
- ▶ **Easier** to distribute the model across devices in an **efficient** way.



RNN Model Parallelization

- ▶ Split the NN **horizontally** by placing **each layer** on a **different device**.
- ▶ At the **first step**, only one device will be active.
- ▶ At the **second step**, two will be active.
- ▶ While the **first layer** will be handling the **second value**, the **second layer** will be handling the output of the **first layer** for the first value.
- ▶ By the time the signal propagates to the **output layer**, all devices will be active **simultaneously**.



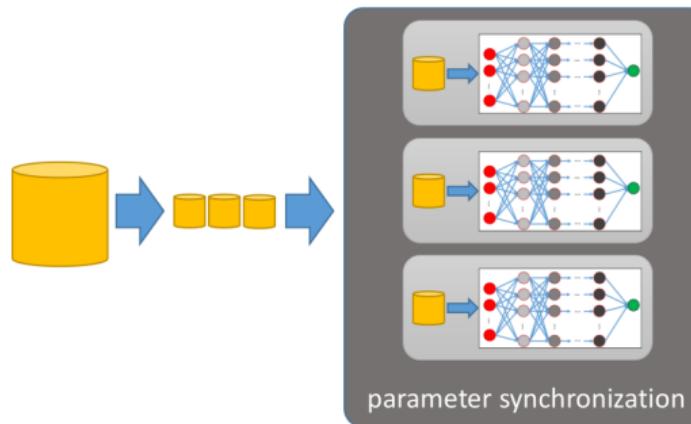
Deep recurrent
neural network



Data Parallelization

Data Parallelization (1/2)

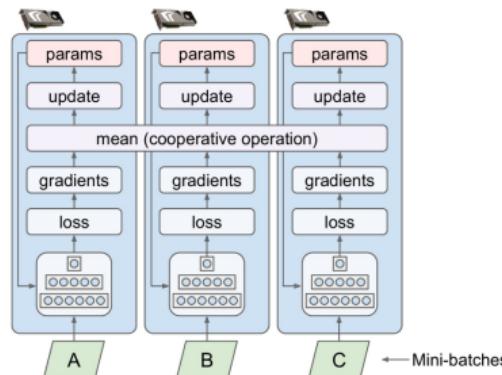
- ▶ Replicate a **whole model** on **every device**.
- ▶ Train **all replicas simultaneously**, using a **different mini-batch** for each.



[Mayer, R. et al., arXiv:1903.11314, 2019]

Data Parallelization (2/2)

1. Compute the **gradient** of the **loss function** using a **mini-batch** on each GPU.
2. Compute the **mean of the gradients** by **inter-GPU communication**.
3. **Update the model**.





Data Parallelization Design Issues

- ▶ System Architecture: how to synchronize the parameters
- ▶ Synchronization: when to synchronize the parameters



System Architecture

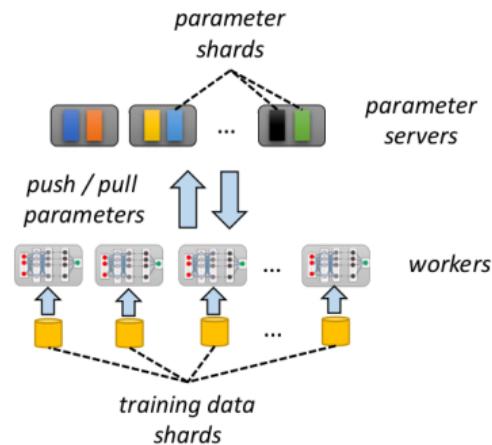
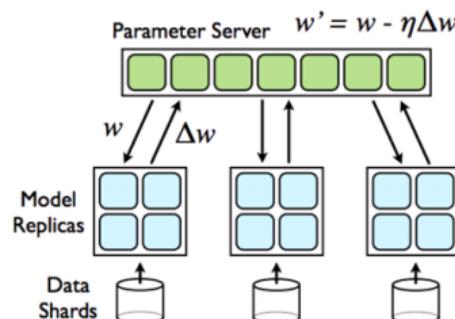


System Architecture - Centralized

- ▶ How to aggregate gradients (compute the mean of the gradients)?
- ▶ How the parameters of the different replicas are synchronized?

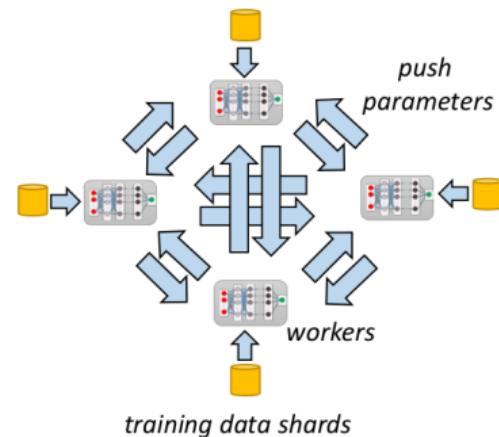
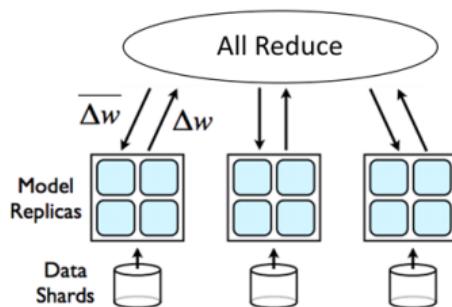
System Architecture - Centralized

- ▶ Store the model parameters **outside of the workers**.
- ▶ **Workers** periodically report their **computed parameters** or **parameter updates** to a (set of) **parameter server(s)** (PSs).



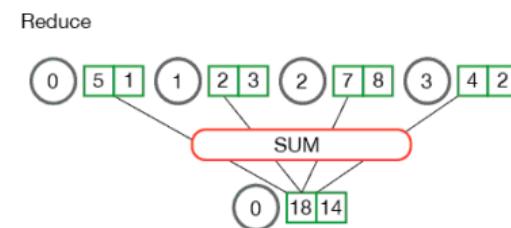
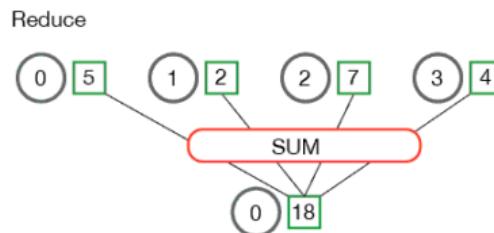
System Architecture - Decentralized

- ▶ Mirror all the model parameters across all workers (No PS).
- ▶ Workers exchange parameter updates directly via an **allreduce** operation.



Reduce and AllReduce (1/2)

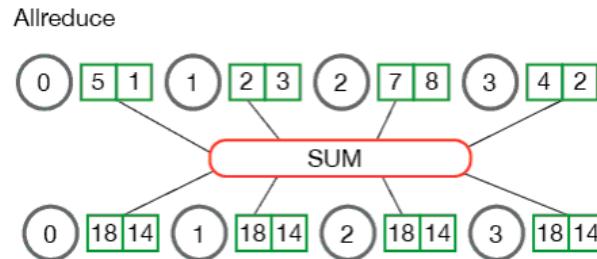
- ▶ **Reduce**: reducing a **set of numbers** into a **smaller set of numbers** via a function.
- ▶ E.g., `sum([1, 2, 3, 4, 5]) = 15`
- ▶ Reduce takes an **array of input** elements on each process and returns an **array of output** elements to the root process.



[<https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce>]

Reduce and AllReduce (2/2)

- ▶ AllReduce stores reduced results across all processes rather than the root process.



[<https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce>]

AllReduce Example

Initial state

Worker A
17 11 1 9

Worker B
5 13 23 14

Worker C
3 6 10 8

Worker D
12 7 2 12



After AllReduce operation

Worker A
37 37 36 43

Worker B
37 37 36 43

Worker C
37 37 36 43

Worker D
37 37 36 43

[<https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da>]

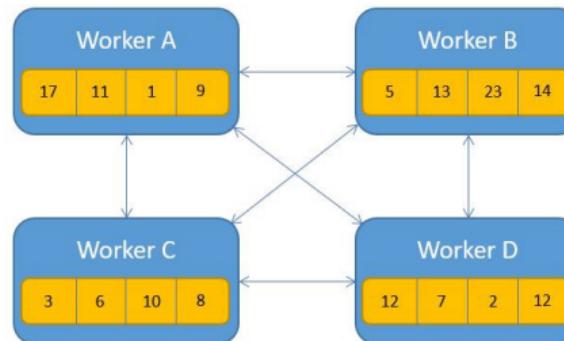


AllReduce Implementation

- ▶ All-to-all allreduce
- ▶ Master-worker allreduce
- ▶ Tree allreduce
- ▶ Round-robin allreduce
- ▶ Butterfly allreduce
- ▶ Ring allreduce

AllReduce Implementation - All-to-All AllReduce

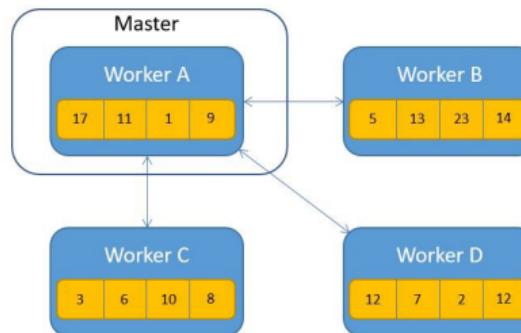
- ▶ Send the array of data to each other.
- ▶ Apply the reduction operation on each process.
- ▶ Too many unnecessary messages.



[<https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da>]

AllReduce Implementation - Master-Worker AllReduce

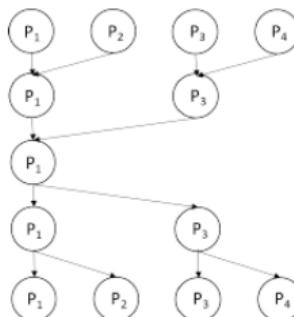
- ▶ Selecting one process as a **master**, gather all arrays into the master.
- ▶ Perform **reduction operations** locally in the **master**.
- ▶ **Distribute the result** to the **other processes**.
- ▶ The master becomes a **bottleneck** (**not scalable**).



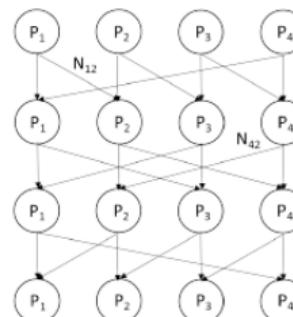
[<https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da>]

AllReduce Implementation - Other implementations

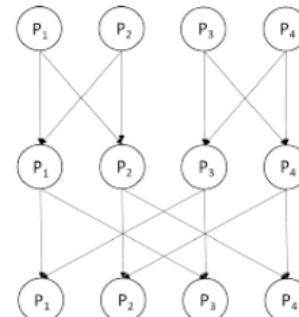
- ▶ Some try to minimize bandwidth.
- ▶ Some try to minimize latency.



(a) Tree AllReduce



(b) Round-robin AllReduce



(c) Butterfly AllReduce

[Zhao H. et al., arXiv:1312.3020, 2013]

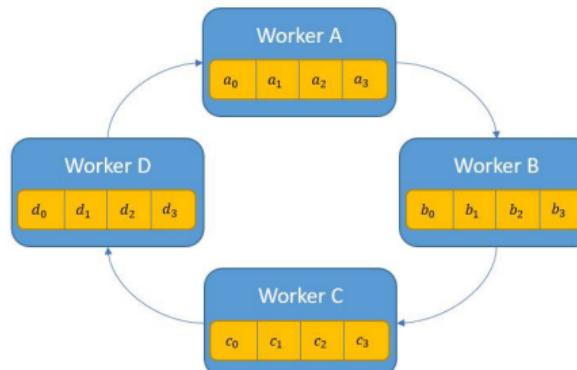


AllReduce Implementation - Ring-AllReduce (1/6)

- ▶ The **Ring-Allreduce** has two phases:
 1. First, the **share-reduce** phase
 2. Then, the **share-only** phase

AllReduce Implementation - Ring-AllReduce (2/6)

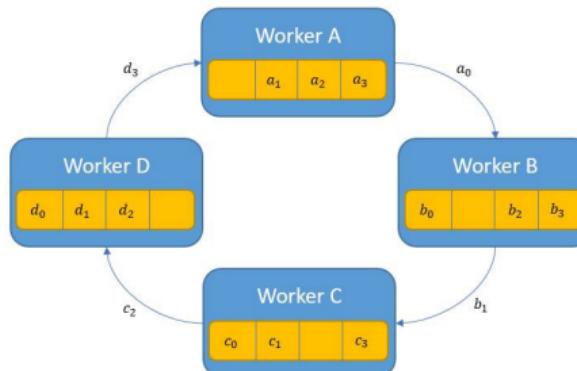
- In the **share-reduce** phase, each process p sends data to the process $(p+1) \% m$
 - m is the number of processes, and $\%$ is the modulo operator.
- The **array of data** on each process is divided to m chunks ($m=4$ here).
- Each one of these **chunks** will be **indexed** by i going forward.



[<https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da>]

AllReduce Implementation - Ring-AllReduce (3/6)

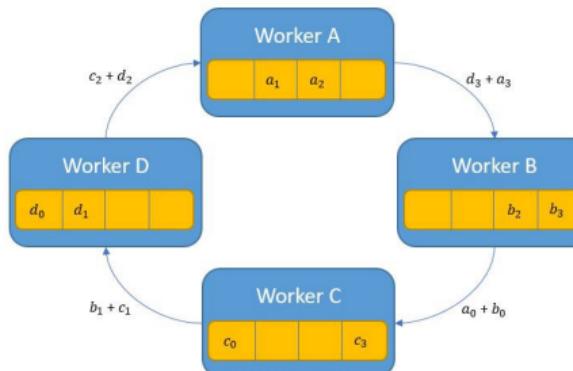
- ▶ In the **first share-reduce step**, process **A** sends **a₀** to process **B**.
- ▶ Process **B** sends **b₁** to process **C**, etc.



[<https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da>]

AllReduce Implementation - Ring-AllReduce (4/6)

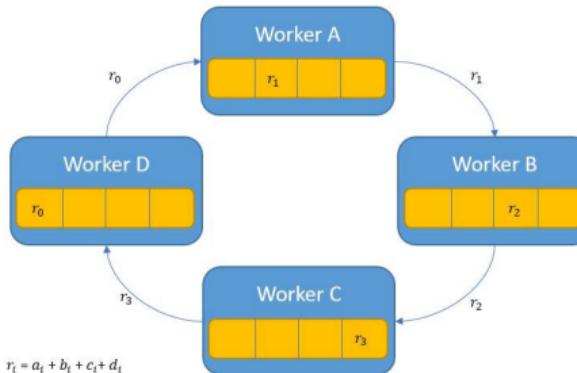
- ▶ When each process **receives the data from the previous process**, it applies the **reduce operator** (e.g., sum or mean)
 - The **reduce operator** should be **associative** and **commutative**.
- ▶ It then proceeds to **send it to the next process in the ring**.



[<https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da>]

AllReduce Implementation - Ring-AllReduce (5/6)

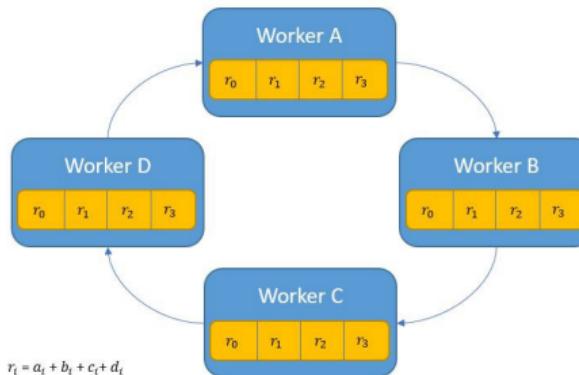
- ▶ The share-reduce phase finishes when each process holds the complete reduction of chunk i.
- ▶ At this point each process holds a part of the end result.



[<https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da>]

AllReduce Implementation - Ring-AllReduce (6/6)

- ▶ The **share-only** step is the same process of sharing the data in a ring-like fashion **without applying the reduce operation**.
- ▶ This **consolidates** the **result of each chunk** in **every process**.



[<https://towardsdatascience.com/visual-intuition-on-ring-allreduce-for-distributed-deep-learning-d1f34b4911da>]

Master-Worker AllReduce vs. Ring-AllReduce

- ▶ N : number of elements, m : number of processes
- ▶ Master-Worker AllReduce
 - First each **process** sends N elements to the **master**: $N \times (m - 1)$ messages.
 - Then the **master** sends the results back to the **process**: another $N \times (m - 1)$ messages.
 - Total network traffic is $2(N \times (m - 1))$, which is **proportional** to m .
- ▶ Ring-AllReduce
 - In the **share-reduce** step each **process** sends $\frac{N}{m}$ elements, and it does it $m - 1$ times:
 $\frac{N}{m} \times (m - 1)$ messages.
 - On the **share-only** step, each **process** sends the result for the chunk it calculated: another
 $\frac{N}{m} \times (m - 1)$ messages.
 - Total network traffic is $2(\frac{N}{m} \times (m - 1))$.



Synchronization

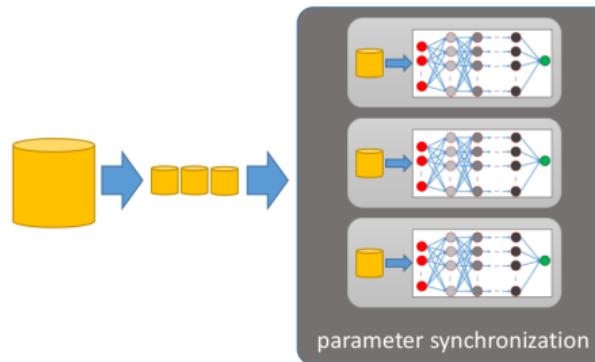


Synchronization

- ▶ When to synchronize the parameters among the parallel workers?

Synchronization - Synchronous

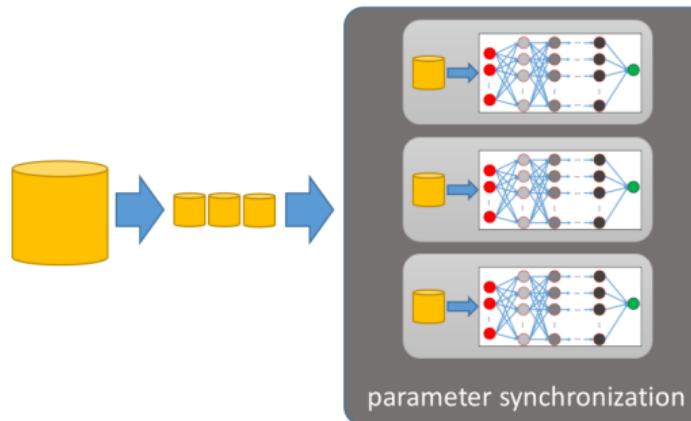
- ▶ After each **iteration** (**processing of a mini-batch**), the workers **synchronize** their parameter updates.
 - Easy to reason about the **model convergence**.
 - The training process **prone to the straggler** problem, where the **slowest** worker **slows down all the others**.



[Mayer, R. et al., arXiv:1903.11314, 2019]

Synchronization - Asynchronous

- ▶ Workers update their model independently from each other.
 - A **worker** may train on **stale (delayed)** parameters.
 - This makes it **hard** to mathematically reason about the **model convergence**.
 - It provides the workers **flexibility** in their training process, completely avoiding all **straggler problems**.



[Mayer, R. et al., arXiv:1903.11314, 2019]



Data Parallelization in TensorFlow



TensorFlow Distribution Strategies

- ▶ `tf.distribute.Strategy` is a TensorFlow API to **distribute training**.
- ▶ Supports both **parameter server** and **allreduce** models.



Single Server



Single Server Training - MirroredStrategy (1/2)

- ▶ Synchronous distribute training training on multiple GPUs on one machine.
- ▶ One replica per GPU.
- ▶ The parameters of the model are mirrored across all the replicas.
- ▶ These parameters are kept in sync with each other by applying identical updates.
- ▶ The parameters updates are communicated using allreduce algorithms.

```
mirrored_strategy = tf.distribute.MirroredStrategy()  
  
# to use only some of the GPUs on your machine  
mirrored_strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```



Single Server Training - MirroredStrategy (2/2)

- ▶ There are different implementation of `allreduce`.
- ▶ You can `override` the cross GPU communication:
 - `tf.distribute.NcclAllReduce` (the default)
 - `tf.distribute.ReductionToOneDevice`
 - `tf.distribute.HierarchicalCopyAllReduce`

```
mirrored_strategy = tf.distribute.MirroredStrategy(  
    cross_device_ops=tf.distribute.HierarchicalCopyAllReduce())
```



Single Server Training - CentralStorageStrategy

- ▶ Parameters are not mirrored, instead they are placed on the CPU.
- ▶ Operations are replicated across all local GPUs.
- ▶ Does synchronous training.

```
central_storage_strategy = tf.distribute.experimental.CentralStorageStrategy()
```



Single Server Trainings - Example

- ▶ Create a **strategy**, e.g., `MirroredStrategy` or `CentralStorageStrategy`.
- ▶ Call its `scope()` method to get a **distribution context**.
- ▶ Wrap the `creation` and `compilation` of the `model` **inside that context**.
- ▶ Call the model's `fit()` and `predict()` method normally (**outside the context**).

```
distribution = tf.distribute.MirroredStrategy()

with distribution.scope():
    model = keras.models.Sequential([...])
    model.compile(...)

model.fit(...)
model.predict(...)
```



Multi Servers



Multi Servers Trainings - MultiWorkerMirroredStrategy (1/2)

- ▶ Very similar to MirroredStrategy.
- ▶ Synchronous distributed training across multiple workers, each with potentially multiple GPUs.
- ▶ Makes copies of all parameters of the model on each device across all workers.

```
multiworker_strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy()
```



Multi Servers Trainings - MultiWorkerMirroredStrategy (2/2)

- ▶ Two different **implementations**:
 - `CollectiveCommunication.RING` (ring-based implementation)
 - `CollectiveCommunication.NCCL` (Nvidia's NCCL implementation)
- ▶ `CollectiveCommunication.AUTO` defers the choice to the **runtime**.
- ▶ The **best choice** of collective implementation depends upon the **number and kind of GPUs**, and the **network interconnect** in the cluster.

```
# ring-based collectives
multiworker_strategy = tf.distribute.experimental.MultiWorkerMirroredStrategy(
    tf.distribute.experimental.CollectiveCommunication.RING)
```



Multi Servers Trainings - ParameterServerStrategy

- ▶ Supports parameter servers training on multiple machines.
- ▶ Some machines are designated as workers and some as parameter servers.
- ▶ Each parameter of the model is placed on one parameter server.
- ▶ Computation is replicated across all GPUs of all the workers.

```
ps_strategy = tf.distribute.experimental.ParameterServerStrategy()
```



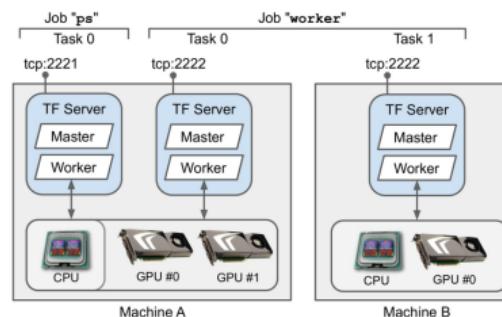
Multi Servers Trainings - More Details

- ▶ A TensorFlow **cluster** is a group of TensorFlow **processes** running in parallel.
- ▶ Each TF **process** (a.k.a **task**) in the cluster has a **type**:
 - **Worker**: performs **computations**, usually on a machine with one or more **GPUs**.
 - **Parameter Server (ps)**: keeps track of **parameters values**, it is usually on a **CPU-only** machine.
- ▶ The **set of tasks** that share the **same type** is often called a **job**. For example, the **worker job** is the **set of all workers**.

Multi Servers Trainings - Example (1/3)

- ▶ Assume a cluster with 3 tasks (2 workers and 1 parameter server).

```
cluster_spec = tf.train.ClusterSpec({  
    "worker": [  
        "machine-a.example.com:2222", # /job:worker/task:0  
        "machine-b.example.com:2222"  # /job:worker/task:1  
    ],  
    "ps": ["machine-a.example.com:2221"] # /job:ps/task:0  
})
```





Multi Servers Trainings - Example (2/3)

- ▶ To start a task, you must give it the `cluster_spec` and define its `type` and `index (ID)`, e.g., worker 0.

```
ps0 = tf.distribute.Server(cluster_spec, job_name="ps", task_index=0)
```

```
worker0 = tf.distribute.Server(cluster_spec, job_name="worker", task_index=0)
```

```
worker1 = tf.distribute.Server(cluster_spec, job_name="worker", task_index=1)
```



Multi Servers Trainings - Example (3/3)

- ▶ Alternative way to specify a `cluster spec` is to use the `TF_CONFIG` environment variable before starting the program.
- ▶ For example to run `worker 1`:

```
distribution = tf.distribute.experimental.ParameterServerStrategy()

os.environ["TF_CONFIG"] = json.dumps({
    "cluster": {
        "worker": ["machine-a.example.com:2222", "machine-b.example.com:2222"],
        "ps": ["machine-a.example.com:2221"]},
    "task": {"type": "worker", "index": 1}
})

with distribution.scope():
    model = keras.models.Sequential([...])
    model.compile(...)

model.fit(...)
```



Communication Overhead



Communication Overhead in Data Parallelization

- ▶ Synchronizing the model replicas in data-parallel training requires communication between workers (in `allreduce`)
- ▶ Between workers and parameter servers (in the centralized architecture).
- ▶ Such communication can easily become the bottleneck of the overall training process.



Approaches for Communication Efficiency

- ▶ Reducing the model precision
- ▶ Compressing the model updates
- ▶ Improving the communication scheduling



Reducing the Model Precision

- ▶ Reduce the precision of the parameters' data types, e.g., from double precision to single floating point.
- ▶ It saves communication bandwidth when parameter updates need to be transferred over the network.
- ▶ It reduces the model size, which can be useful when the model is deployed on resource-constrained hardware such as GPUs.



Compressing the Model Updates

- ▶ The **model updates** communicated between workers and between workers and parameter servers can be compressed.
- ▶ **Gradient quantization:** reducing the number of bits per gradient.
- ▶ **Gradient sparsification:** communicating only important gradients that have a significant value.



Improving the Communication Scheduling

- ▶ Communication patterns in data-parallel are typically bursty, especially in synchronous systems.
 - All workers may share their updated parameters at the same time with their peer workers or parameter servers.
- ▶ To prevent that the network bandwidth is exceeded and communication is delayed, the communication of the different workers can be scheduled such that it does not overlap.
 - Prioritize specific messages over others.



Summary



Summary

- ▶ CPU vs. GPU
- ▶ Parallelization
- ▶ Model-parallel
- ▶ Data-parallel
 - Parameter server vs. AllReduce
 - Synchronized vs. asynchronous
- ▶ Communication challenges



Reference

- ▶ Aurélien Géron, Hands-On Machine Learning (Ch. 19)
- ▶ Mayer, R. et al., “Scalable Deep Learning on Distributed Infrastructures: Challenges, Techniques and Tools”, 2019.



Questions?